A Database for Dynamic Distributed Content
and its Application for Service and Resource Discovery

Wolfgang Hoschek
CERN – European Organization for Nuclear Research
CH-1211 Geneva 23, Switzerland
Wolfgang.Hoschek@cern.ch

Abstract

In a distributed system such as a DataGrid, it is often desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. This enables information discovery and collective collaborative functionality that operate on the system as a whole, rather than on a given part of it. However, it is not obvious how a database (registry) should maintain information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. In particular, how can one avoid sacrificing reliability, predictability and simplicity while allowing to express powerful queries over time-sensitive dynamic information? We propose the so-called *hyper registry*, which has has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client.

1 Introduction

The next generation Large Hadron Collider (LHC) project at CERN, the European Organization for Nuclear Research, involves thousands of researchers and hundreds of institutions spread around the globe. A massive set of computing resources is necessary to support it's data-intensive physics analysis applications, including thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [1]. To make collaboration viable, it was decided to share in a global joint effort - the European DataGrid (EDG) [2, 3, 4, 5] - the data and locally available resources of all participating laboratories and university departments.

Grid technology attempts to support flexible, secure, coordinated information sharing among dynamic collections of individuals, institutions and resources. This includes data

sharing but also includes access to computers, software and devices required by computation and data-rich collaborative problem solving [6]. These and other advances of distributed computing are necessary to increasingly make it possible to join loosely coupled people and resources from multiple organizations.

An enabling step towards increased Grid software execution flexibility is the (still immature and hence often hyped) *web services* vision [2, 7, 8] of distributed computing where programs are no longer configured with static information. Rather, the promise is that programs are made more flexible and powerful by consulting Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components. For example, a data-intensive High Energy Physics analysis application sweeping over Terabytes of data looks for remote services that exhibit a suitable combination of characteristics, including network load, available disk quota, access rights, and perhaps Quality of Service and monetary cost.

More generally, in a distributed system, it is often desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. This enables information discovery and collective collaborative functionality that operate on the system as a whole, rather than on a given part of it. For example, it allows a search for descriptions of services of a file sharing system, to determine its total download capacity, the names of all participating organizations, etc. Example systems include a service discovery system for a (worldwide) DataGrid, an electronic market place, or an instant messaging and news service. In all these systems, a variety of information describes the state of autonomous remote participants residing within different administrative domains. Participants frequently join, leave and act on a best effort basis. In a large distributed system spanning many administrative domains, predictable, timely, consistent and reliable global state maintenance is infeasible. The information to be aggregated and integrated may be outdated, inconsistent, or not available at all. Failure, misbehavior, security restrictions and continuous change are the norm rather than the exception. In this paper, a design and specification for a type of database is developed that is conditioned to address the problem, the so-called *hyper registry*. A hyper registry has a database that holds a set of tuples. A *tuple* may contain a piece of *content*. Examples for content include a service description expressed in WSDL [9], a file, picture, current network load, host information, stock quotes, etc., as depicted in Figure 1.

This paper is organized as follows. Section 2 and 3 discuss content description and publication. In a given context, a content provider can publish content of a given type to one or more registries. More precisely, a *content provider* can publish a dynamic pointer called a *content link*, which in turn enables the hyper registry and third parties to retrieve (pull) the current content presented from the provider at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push). In any case, a hyper registry may support caching of content.

Section 4 illustrates how a remote client can query a hyper registry, obtaining a set of tuples as answer. The use of XQuery [10, 11, 12, 13] as a rich and expressive query language is motivated and discussed by means of examples.

Section 5 proposes solutions to the arising cache coherency issues. Content *caching* is

important for client efficiency. The hyper registry may not only keep links but also a copy of the current content pointed to by the link. With caching, clients no longer need to establish a network connection for each content link in a query result set in order to obtain content. This avoids prohibitive latency, in particular in the presence of large result sets. A hyper registry may (but need not) support caching, for example in server pull or client push mode or both.

For reliable, predictable and simple state maintenance, Section 6 proposes to keep a hyper registry tuple as soft state. A tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications from the content provider. To this end, a tuple carries timestamps. A tuple is expired and removed unless explicitly renewed via timely periodic publication, henceforth termed *refresh*. In other words, a refresh allows a content provider to cause a content link and/or cached content to remain present for a further time.

Section 7 shows that content link, content cache, a hybrid pull/push communication model and the expressive power of XQuery allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client.

Section 8 compares our approach with related work. Section 9 summarizes and concludes this paper.

2 Content Link and Content Provider

Content Link. A *content link* is an HTTP(S) URL pointing to the content of a content provider. An HTTP(S) GET request to the (universally unique) link must return the current content. In other words, a simple hyperlink is employed. In the context of service discovery, we use the term *service link* to denote a content link that points to a service description. Like in the WWW, content links can freely be chosen as long as they conform to the HTTP URL specification [14]. Examples for legal content links are:

```
http://sched001.cern.ch:8080/presenter/getServiceDescription.wsdl
https://svrpub.cern.ch/getServiceDescription?id=4712&cache=disable
```

Content Provider. A *content provider* offers information conforming to a homogeneous global data model. In order to do so, it typically uses some kind of internal mediator to transform information from a local or proprietary data model to the global data model. A content provider can be seen as a gateway to heterogeneous content sources. The global data model is the Dynamic Data Model (DDM) introduced in Section 3. Content can be structured or semi-structured data in the form of any arbitrary well-formed XML [15] document or fragment. Individual content may, but need not, have a schema (XML Schema [16]), in which case content must be valid according to the schema. All content may, but need not, share a common schema. This flexibility is important for integration of heterogeneous content.

A content provider is an umbrella term for two components, namely a presenter and a publisher. The *presenter* is a service and answers HTTP(S) GET content retrieval requests from a hyper registry or client (subject to local security policy). The *publisher* is a piece of code that publishes content link, and perhaps also content, to a hyper registry. The publisher

need not be a service, although it uses HTTP(S) POST for transport of communications. The structure of a content provider and its interaction with a hyper registry and a client are depicted in Figure 2 (a). Note that a client can bypass a hyper registry and directly pull current content from a provider. Figure 2 (b) illustrates a hyper registry with several content providers and clients.

Just as in the dynamic WWW that allows for a broad variety of implementations for the given protocol, it is left unspecified how a presenter computes content on retrieval. Content can be static or dynamic (generated on the fly). For example, a presenter may serve the content directly from a file or database, or from a potentially outdated cache. For increased accuracy, it may also dynamically recompute the content on each request. Consider the example providers in Figure 3. A simple but nonetheless very useful content provider uses a commodity HTTP server such as Apache to present XML content from the file system. A simple `cron` job monitors the health of the Apache server and publishes the current state to a hyper registry. Another example for a content provider is a Java servlet that makes available data kept in a relational or LDAP database system. A content provider can execute legacy command line tools to publish system state information such as network statistics, operating system and type of CPU. Another example for a content provider is a network service such as a replica catalog that (in addition to servicing replica lookup requests) publishes its service description and/or link so that clients may discover and subsequently invoke it.

Providers and registries can be deployed and configured arbitrarily. For example, in a strategy for scalable administration of large cluster environments, a single shared Apache web server can easily be configured to serve XML descriptions of thousands of services on hundreds of machines. For example, via a naming convention we can assign a distinct web server directory and corresponding service link for each host-service combination. To serve descriptions it is sufficient to have some administrative `cron` job run periodically on each service machine, which writes the current service description into an appropriate XML file in the appropriate directory on the web server.

3 Publication

In a given context, a content provider can publish content of a given type to one or more registries. More precisely, a content provider can publish a dynamic pointer called a content link, which in turn enables the hyper registry (and third parties) to retrieve the current content. For efficiency, the `publish` operation takes as input a set of zero or more tuples. In what we propose to call the *Dynamic Data Model (DDM)*, each XML tuple has a content link, a type, a context, some soft state time stamps, and (optionally) metadata and content. A tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary *content* and allows for refresh of that content at any time. Consider the dynamic tuple set depicted in Figure 4.

- **Link.** The content link is an HTTP(S) URL as introduced above. Given the link the current content of a content provider can be retrieved (pulled) at any time.

- **Type.** The type describes *what* kind of content is being published (e.g. `service`, `application/octet-stream`, `image/jpeg`, `networkLoad`, `hostinfo`).

```
<service>
  <interface type = "http://gridforum.org/interface/scheduler-1.0">
    <operation>
      <name>void submitJob(String jobdescription)</name>
      <allow> http://cms.cern.ch/everybody </allow>
      <bind:http verb="GET" URL="https://sched.cern.ch/scheduler/submitjob"/>
    </operation>
  </interface>
</service>
```

```
<hostInfo>
   <host name="fred01.cern.ch" os="redhat 7.2" arch="i386" MHz="1000" cpus="2"/>
   <host name="fred02.cern.ch" os="solaris 2.7" arch="sparc" MHz="400" cpus="64"/>
</hostInfo>
```

Figure 1: Example Content.

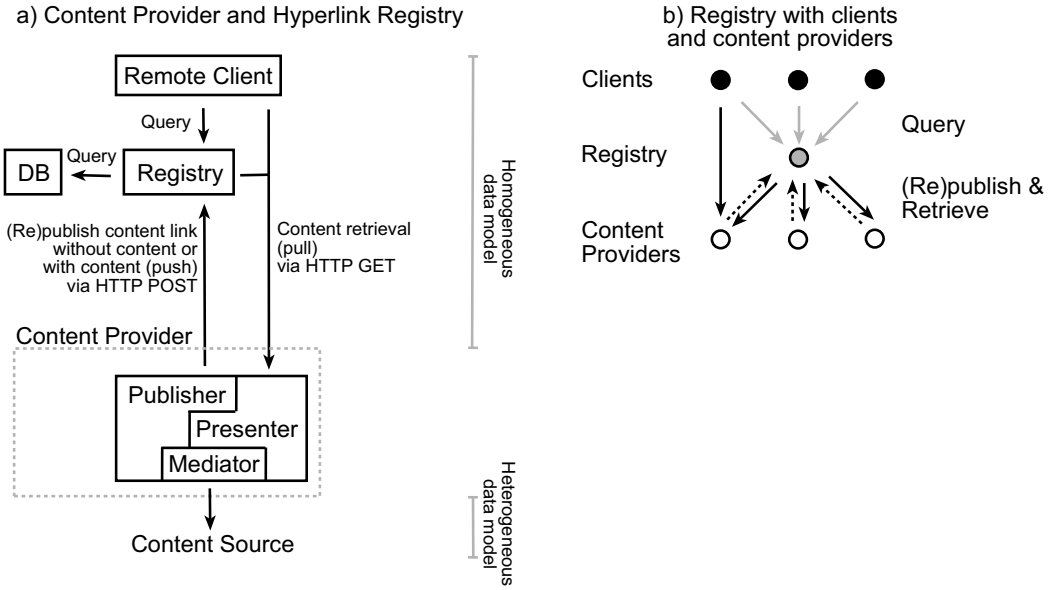

Figure 2: Content Provider and Hyper Registry.

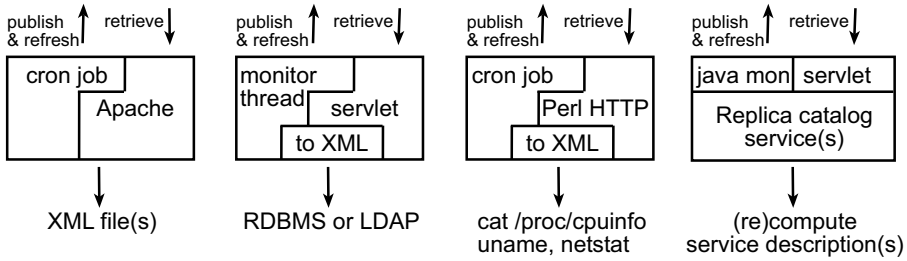

Figure 3: Example Content Providers.

- **Context.** The context describes *why* the content is being published or *how* it should be used (e.g. `child, parent, x-ireferral, gnutella, monitoring`). Context and type allow a query to differente on crucial attributes even if content caching is not supported or not authorized.

- **Timestamps TS1, TS2, TS3.** Discussion of timestamps is deferred to Section 6 below.

- **Metadata.** The metadata element further describes the content and/or its retrieval beyond what can be expressed with the previous attributes. For example, the metadata may be a secure digital signature of the content. It may describe the authoritative content provider or owner of the content. Another metadata example is a Web Service Inspection Language (WSIL) document [17] or fragment thereof, specifying additional content retrieval mechanisms beyond HTTP content link retrieval. The metadata argument may be any well-formed XML document or fragment. It is an extensibility element enabling customization and flexible evolution. It is optional.

- **Content.** Given the link the current content of a content provider can be retrieved (pulled) at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push). For clarity of exposition, the published content is an XML string[1].

The publish operation of a hyper registry has the signature `void publish(XML tupleset)`. A tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. If a key does not already exist on publication, a tuple is inserted into the hyper registry database. An existing tuple can be updated by publishing other values under the same tuple key. An existing tuple (key) is "owned" by the content provider that created it with the first publication. It is recommended that a content provider with another identity may not publish or update the tuple.

4 Query

Minimalist Query. As a minimum, clients can query the hyper registry by invoking minimalist query operations such as `getTuples`. The `getTuples` query operation takes no arguments and returns the full set of all published tuples "as is". That is, query output format and publication input format are the same (see Figure 4). If supported, output includes cached content. The `getLinks` query operation is similar in that it also takes no arguments and returns the full set of all tuples. However, it always substitutes an empty string for cached content. In other words, the content is omitted from tuples, potentially saving substantial bandwidth. The second tuple in Figure 4 has such a form.

[1]In the general case (allowing non-text based content types such as `image/jpeg`), the content is a MIME object. The XML based publication input set and query result set is augmented with an additional MIME multipart object [18], which is a list containing all content. The content element of the result set is interpreted as an index into the MIME multipart object. A typical hyper registry that supports caching can handle content with at least MIME content-type `text/xml` and `text/plain`.

```
<tupleset>
    <tuple link="http://sched001.cern.ch/getServiceDescription"
           type="service" ctx="parent" TS1="10" TC="15" TS2="20" TS3="30">
       <content>    <service> service description A goes here </service>    </content>
       <metadata>  <owner name="http://cms.cern.ch"/>  </metadata>
    </tuple>
    <tuple link="http://repcat.cern.ch/pub/getServiceDescription?id=4711"
           type="service" ctx="child" TS1="30" TC="0" TS2="40" TS3="50">
    </tuple>
</tupleset>
```

Figure 4: Dynamic Tuple Set.

- *Find all (available) services.*

```
RETURN /tupleset/tuple[@type="service"]
```

- *Find all services that implement a replica catalog service interface and that CMS members are allowed to use, and that have an HTTP bindings for the replica catalog operation "XML getPFNs(String LFN)".*

```
LET $repcat := "http://gridforum.org/interface/replicaCatalog-1.0"
FOR $tuple IN /tupleset/tuple[@type="service"]
WHERE SOME $op IN $tuple/content/service/interface[@type = $repcat]/operation
      SATISFIES ($op/name="XML getPFNs(String LFN)" AND $op/bindhttp/@verb="GET"
                 AND contains($op/allow, "http://cms.cern.ch/everybody"))
RETURN $tuple
```

- *Return the number of replica catalogs.*

```
LET $repcat := "http://gridforum.org/interface/replicaCatalog-1.0"
RETURN count(/tupleset/content/service[interface/@kind=$repcat])
```

- *Find all (execution service, storage service) pairs where both services of a pair live within the same domain. (Job wants to read and write locally).*

```
LET $exeType := "http://gridforum.org/interface/executor-1.0"
LET $stoType := "http://gridforum.org/interface/storage-1.0"
FOR $executor IN/tupleset/tuple[content/service/interface/@type = $exeType],
    $storage  IN /tupleset/tuple[content/service/interface/@type = $stoType AND
    domainName(@link) = domainName($executor/@link)]
RETURN <pair> {$executor} {$storage} </pair>
```

Figure 5: Simple, Medium and Complex XQueries for Service Discovery.

XQuery. A hyper registry node has the capability to execute XQueries over the set of tuples it holds in its database. XQuery [10, 11, 12, 13] is the standard XML [15] query language developed under the auspices of the W3C. It allows for powerful searching, which is critical for non-trivial applications. Everything that can be expressed with SQL [19] can also be expressed with XQuery. However, XQuery is a more expressive language than SQL. Simple, medium and complex XQueries for service discovery are depicted in Figure 3. XQuery can integrate external data sources via the `document(URL)` function. The `document(URL)` function can be used to process the XML results of remote operations invoked over HTTP. For example, given a service description with a `getNetworkLoad()` operation, a query can match on values dynamically produced by that operation. For a detailed discussion of a wide range of discovery queries, their representation in the XQuery language, as well as detailed motivation and justification, see our prior studies [2]. The same rules that apply to minimalist queries also apply to XQuery support. An implementation can use a modular and simple XQuery processor such as `Quip` [20] for an operation such as `XML query(XQuery query)`. Because not only content, but also content link, context, type, time stamps, metadata etc. are part of a tuple, a query can also select on this information.

5 Caching

Content *caching* is important for client efficiency. The hyper registry may not only keep content links but also a copy of the current content pointed to by the link. With caching, clients no longer need to establish a network connection for each content link in a query result set in order to obtain content. This avoids prohibitive latency, in particular in the presence of large result sets. A hyper registry may (but need not) support caching. A hyper registry that does not support caching ignores any content handed from a content provider. It keeps content links only. Instead of cached content it returns empty strings. Cache coherency issues arise. The query operations of a caching hyper registry may return tuples with stale content, i.e. content that is out of date with respect to its master copy at the content provider.

A caching hyper registry may implement a *strong* or *weak cache coherency policy*. A strong cache coherency policy is *server invalidation* [21]. Here a content provider notifies the hyper registry with a publication message whenever it has locally modified the content. We use this approach in an adapted version where a caching hyper registry can operate according to the client push pattern (*push hyper registry*) or server pull pattern (*pull hyper registry*) or a hybrid thereof. The respective interactions are as follows:

- **Pull Hyper Registry.** A content provider publishes a content link. The hyper registry then pulls the current content via content link retrieval into the cache. Whenever the content provider modifies the content, it notifies the hyper registry with a publication message carrying the time the content was last modified. The hyper registry may then decide to pull the current content again, in order to update the cache. It is up to the hyper registry to decide if and when to pull content. A hyper registry may pull content at any time. For example, it may dynamically pull fresh content for tuples affected by a query. This is important for frequently changing dynamic data such as network load.

- **Push Hyper Registry.** A publication message pushed from a content provider to the hyper registry contains not only a content link but also its current content. Whenever a content provider modifies content, it pushes the new content to the hyper registry, which may update the cache accordingly.

- **Hybrid Hyper Registry.** A hybrid hyper registry implements both pull and push interactions. If a content provider merely notifies that its content has changed, the hyper registry may choose to pull the current content into the cache. If a content provider pushes content, the cache may be updated with the pushed content. This is the type of hyper registry subsequently assumed whenever a caching hyper registry is discussed.

A non-caching hyper registry ignores content elements, if present. A publication is said to be *without content* if the content is not provided at all. Otherwise, it is said to be *with content*. Publication without content implies that no statement at all about cached content is being made (neutral). It does *not* imply that content should not be cached or invalidated.

6 Soft State

For reliable, predictable and simple distributed state maintenance, a hyper registry tuple is maintained as *soft state*. A tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications from a content provider. To this end, a tuple carries timestamps. A tuple is expired and removed unless explicitly renewed via timely periodic publication, henceforth termed *refresh*. In other words, a refresh allows a content provider to cause a content link and/or cached content to remain present a for a further time.

The strong cache coherency policy *server invalidation* is extended. For flexibility and expressiveness, the ideas of the Grid Notification Framework [22] are adapted. The publication operation takes three absolute time stamps `TS1, TS2, TS3` per tuple. The semantics are as follows. The content provider asserts that its content was last modified at time `TS1` and that its current content is expected to be valid from time `TS1` until at least time `TS2`. It is expected that the content link is alive between time `TS1` and at least time `TS3`. Time stamps must obey the constraint `TS1` $\leq$ `TS2` $\leq$ `TS3`. `TS2` triggers expiration of cached content, whereas `TS3` triggers expiration of content links. Usually, `TS1` equals the time of last modification or first publication, TS2 equals `TS1` plus some minutes or hours, and `TS3` equals `TS2` plus some hours or days. For example, `TS1`, `TS2` and `TS3` can reflect publication time, 10 minutes, and 2 hours, respectively. A hyper registry tuple carries a timestamp `TC` per tuple that indicates the time when the content cache was last modified internal to the hyper registry (not at the content provider). A hyper registry not supporting caching always has `TC` set to zero. Timestamp semantics can be summarized as follows:

```
TS1 = Time content provider last modified content
TC  = Time hyper registry last modified content cache
TS2 = Expected time while current content at provider is at least valid
TS3 = Expected time while content link at provider is at least valid (alive)
```

For example, a highly dynamic network load provider may publish its link without content and `TS1=TS2` to suggest that it is inappropriate to cache its content. Constants are published with content and `TS2=TS3=infinity`.

Insert, update and delete of tuples occur at the timestamp-driven state transitions summarized in Figure 6. A tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. A tuple can be in one of three states: *unknown*, *not cached*, or *cached*. A tuple is unknown if it is not contained in the hyper registry (i.e. its key does not exist). Otherwise, it is known. When a tuple is assigned *not cached* state, its last internal modification time `TC` is (re)set to zero and the cache is deleted, if present. For a *not cached* tuple we have `TC < TS1`. When a tuple is assigned *cached* state, the content is updated and `TC` is set to the current time. For a *cached* tuple, we have $\texttt{TC} \geq \texttt{TS1}$.

A tuple moves from *unknown* to *cached* or *not cached* state if the provider publishes with or without content, respectively. A tuple becomes *unknown* if its content link expires (`currentTime > TS3`); the tuple is then deleted. A provider can force tuple deletion by publishing with `currentTime > TS3`. A tuple is upgraded from *not cached* to *cached* state if a provider push publishes with content or if the hyper registry pulls the current content itself via retrieval. On content pull, a hyper registry may leave `TS2` unchanged, but it may also follow a policy that extends the lifetime of the tuple (or any other policy it sees fit). A tuple is degraded from *cached* to *not cached* state if the content expires. Such expiry occurs when no refresh is received in time (`currentTime > TS2`), or if a refresh indicates that the provider has modified the content (`TC < TS1`).

7 Flexible Freshness

Content link, content cache, a hybrid pull/push communication model and the expressive power of XQuery allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client. All three components may indicate how to manage content according to their respective notions of freshness. For example, a content provider can model the freshness of its content via pushing appropriate timestamps and content. A hyper registry can model the freshness of its content via controlled acceptance of provider publications and by actively pulling fresh content from the provider. If a result (e.g. network statistics) is up to date according to the hyper registry, but out of date according to the client, the client can pull fresh content from providers as it sees fit. However, this is inefficient for large result sets. Nevertheless, it is important for clients that query results are returned according to their notion of freshness, in particular in the presence of frequently changing dynamic content.

Recall that it is up to the hyper registry to decide to what extent its cache is stale, and if and when to pull fresh content. For example, a hyper registry may implement a policy that dynamically pulls fresh content for a tuple whenever a query touches (affects) the tuple. For example, if a query interprets the content link URL as an identifier within a hierarchical name space (e.g. as in LDAP) and selects only tuples within a sub-tree of the name space, only these tuples should be considered for refresh.

Refresh-on-client-demand. So far, a hyper registry must guess what a client's notion of freshness might be, while at the same time maintaining its decisive authority. A client still has no way to indicate (as opposed to force) its view of the matter to a hyper registry. We propose to address this problem with a simple and elegant *refresh-on-client-demand* strategy under control of the hyper registry's authority. The strategy exploits the rich expressiveness and dynamic data integration capabilities of the XQuery language. The client query may itself inspect the time stamp values of the set of tuples. It may then decide itself to what extent a tuple is considered interesting yet stale. If the query decides that a given tuple is stale (e.g. if `type="networkLoad" AND TC < currentTime() - 10`), it calls the XQuery `document(URL contentLink)` function with the corresponding content link in order to pull and get handed fresh content, which it then processes in any desired way.

This mechanism makes it unnecessary for a hyper registry to guess what a client's notion of freshness might be. It also implies that a hyper registry does not require complex logic for query parsing, analysis, splitting, merging, etc. Moreover, the fresh results pulled by a query can be reused for subsequent queries. Since the query is executed within the hyper registry, the hyper registry may implement the `document` function such that it not only pulls and returns the current content, but as a side effect also updates the tuple cache in its database. A hyper registry retains its authority in the sense that it may apply an authorization policy, or perhaps ignore the query's refresh calls altogether and return the old content instead. The refresh-on-client-demand strategy is simple, elegant and controlled. It improves efficiency by avoiding overly eager refreshes typically incurred by a guessing hyper registry policy.

8 Related Work

RDBMS. The relational data model does not allow for semi-structured data. A query must have out-of-band knowledge of the relevant table names and schemas, which themselves may not be heterogeneous but must be static, globally standardized and synchronized. This seriously limits the applicability of the relational model in the context of autonomy, decentralization, unreliability and frequent change. SQL [19] is a rich and expressive general-purpose language defined over the relational data model. In addition to the above limitations, SQL lacks hierarchical navigation as a key feature and other capabilities such as dynamic data integration, leading to extremely complex queries over a large number of auxiliary tables [23]. Relational database systems do not support an XML data model and the XQuery language. Further, they do not provide soft state based publication and content caching. Content freshness is not addressed. Our work does not compete with an RDBMS, though. A hyper registry may well internally use an RDBMS for data management. The relational data model and SQL are, for example, used in the Relational Grid Monitoring Architecture (RGMA) system [24] and the Unified Relational GIS Project [25].

UDDI. UDDI (Universal Description, Discovery and Integration) [26] is an emerging industry standard that defines a business oriented access mechanism to a registry holding XML based WSDL [9] service descriptions. It is not designed to be a registry holding arbitrary content. UDDI is not based on soft state, which implies that there is no way to dynamically manage and remove service descriptions from a large number of autonomous third parties in

a reliable, predictable and simple way. It does not address the fact that services often fail or misbehave or are reconfigured, leaving a registry in an inconsistent state. Content freshness is not addressed. As such, UDDI only appears to be useful for businesses and their customers running static high availability services. Last, and perhaps most importantly, query support is rudimentary. Only key lookups with primitive qualifiers are supported, which is insufficient for realistic service discovery use cases (see Figure 3 for examples).

SDS. The Service Discovery Service (SDS) [27] is interesting in that it mandates secure channels with authentication and traffic encryption, and privacy and authenticity of service descriptions. SDS servers can be organized in a distributed hierarchy. For efficiency, each SDS node in a hierarchy can hold an index of the content of its sub-tree. The index is a compact aggregation and custom tailored to the narrow type of query SDS can answer. SDS is based on multi cast and soft state. Content freshness is not addressed. It supports a simple XML based exact match query type and hence cannot express the query use cases.

LDAP. The Lightweight Directory Access Protocol (LDAP) [28] defines an access mechanism in which clients send requests to and receive responses from LDAP servers. The database is not based on soft state. Content freshness is not addressed. LDAP does not follow an XML data model. An LDAP query is an expression that logically compares (`=, <=, >=`) the string value of an attribute (`email`) with a string constant, optionally with a substring match joker (`picture*.jpg`) and approximate string equality test ($\sim$). Expressions can be combined with Boolean `AND, OR` and `NOT` operators. The expressive power of the LDAP query language is insufficient for service discovery use cases (see Figure 3 for examples) and most other non-trivial questions.

MDS. The Metacomputing Directory Service (MDS) [29, 30] is based on LDAP. As a result, its query language is insufficient for service discovery, and it does not follow an XML data model. MDS is based on soft state but it does not allow clients (and to some extent even content providers) to drive registry freshness policies.

The MDS consists of an unmodified OpenLDAP [31] server with value-adding backends, configured with a strong security library. The Grid Resource Information Service (GRIS) is an OpenLDAP backend that accepts LDAP queries from clients over its own LDAP namespace sub-tree. It is a backend into which a list of content providers can be plugged on a per attribute basis. An example content provider is a shell script or program that returns the operating system version. A provider owns a namespace sub tree of the GRIS and returns a set of LDAP entries within that namespace. Depending on the namespace specified in an LDAP query, the GRIS executes (and creates the processes for) one or more affected providers and caches the results for use in future queries. It then applies the query against the cache. A GRIS can be statically configured to cache the pulled content for a fixed amount of time (on a per provider basis). An example GRIS configuration invokes the `/usr/local/bin/grid-info-cpu-linux` executable and caches CPU load results for 15 seconds. Content provider invocation follows a CGI like life cycle model. The stateless nature, heavy weight process forking and context switches of such a model render it unsuitable for use in dynamic environments with high frequency refreshes and requests [32].

Figure 7 contrasts the different architectures of a GRIS and a hyper registry. A hyper registry maintains content links and cached content in its database, whereas a GRIS maintains cached content only. The control paths from client to content provider and from content provider to the hyper registry are missing in the GRIS architecture, disabling cache freshness steering. A GRIS content provider is always local to the GRIS and cannot publish to a remote GRIS. In contrast, a content provider is cleanly decoupled from a hyper registry and only requires the ubiquitous HTTP protocol for simple communication with a local or remote registry. A GRIS requires implementing the complex LDAP protocol, including its query language, at every content provider location. In contrast, handling the powerful but complex XQuery language is only required within a hyper registry, not at the content provider. Table 1 summarizes some commonalities and differences related to publication and content freshness.

9 Conclusions

We address the problems of maintaining and querying dynamic and timely information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. The hyper registry has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client. A content provider is cleanly decoupled from the hyper registry and only requires the ubiquitous HTTP protocol for communication with a local or remote hyper registry.

These key properties distinguish our approach from related work, which individually addresses some, but not all of the above issues. Some work does not follow an XML data model (X.500, LDAP, MDS, RDBMS). Sometimes the query language is not powerful enough (UDDI, X.500, LDAP, MDS, SDS). Sometimes the database is not based on soft state (RDBMS, UDDI, X.500, LDAP). Sometimes content freshness is not addressed (RDBMS, UDDI, X.500, LDAP, SDS) or only partly addressed (MDS).

As future work, it would be interesting to study and specify in more detail specific cache freshness interaction policies between content provider, hyper registry and client (query). Our specification allows expressing a wide range of policies, some of which we outline, but we do not evaluate in detail the merits and drawbacks of any given policy.

References

[1] Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001. http://lhc-computing-review-public.web.cern.ch/lhc-computing-review-public/Public/Report_final.PDF.

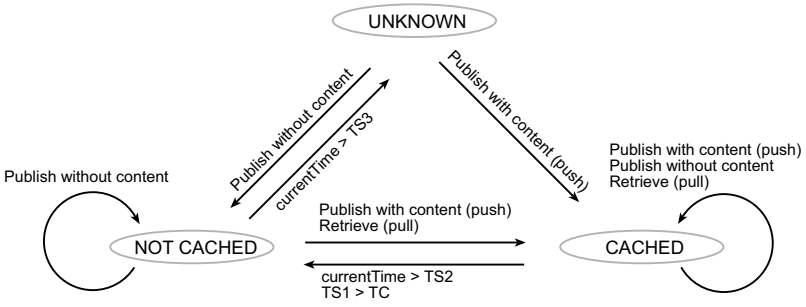

Figure 6: Soft State Transitions.

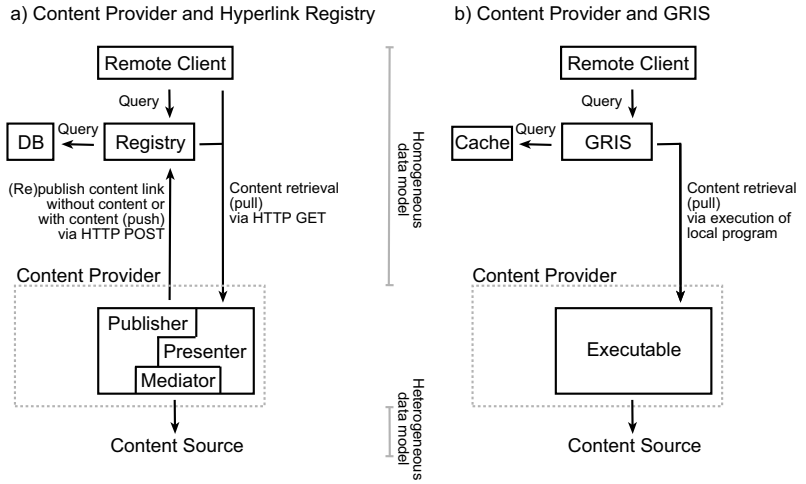

Figure 7: Hyper Registry and Grid Resource Information Service.

Question	**MDS**	**Hyper Registry**
Can a provider publish to a remote registry?	No. A provider is local to a GRIS.	Yes. A provider can publish to any hyper registry, no matter whether the hyper registry is deployed locally, remotely or in-process.
Can a provider actively steer registry freshness?	No. A GRIS is actively pulling content. It can be statically configured to cache the pulled content for a fixed amount of time (on a per provider basis). A content provider is passive. It cannot actively publish and refresh content and hence cannot steer the freshness of its content cached in the GRIS.	Yes. Content provider and hyper registry are active and passive at the same time. At any time, a hyper registry can actively pull content, and a content provider can actively push with or without content. Both components can steer the freshness of content cached in the hyper registry.
Can a client retrieve current content?	No. A client cannot retrieve the current content from a content provider. It has to go through a GRIS or GIIS, which normally return stale content from their cache.	Yes. A client can directly connect to a content provider and retrieve the current content, thereby avoiding stale content from a hyper registry.
Can a client query steer result freshness?	No. A client query cannot steer the freshness of the results it generates.	Yes. A client query can steer the freshness of the results it generates via the refresh-on-client-demand strategy.

Table 1: Comparison of Hyper Registry and Metacomputing Directory Service.

[2] Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery*. PhD Thesis, Technical University of Vienna (submitted), 2002.

[3] Ben Segal. Grid Computing: The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.

[4] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.

[5] Dirk Düllmann, Wolfgang Hoschek, Javier Jean-Martinez, Asad Samar, Ben Segal, Heinz Stockinger, and Kurt Stockinger. Models for Replica Synchronisation and Consistency in a Data Grid. In *10th IEEE Symposium on High Performance and Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.

[6] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. Journal of Supercomputer Applications*, 15(3), 2001.

[7] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002.

[8] P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev. *Professional XML Web Services*. Wrox Press, 2001.

[9] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. *W3C Note 15*, 2001. www.w3.org/TR/wsdl.

[10] World Wide Web Consortium. XQuery 1.0: An XML Query Language. *W3C Working Draft*, December 2001.

[11] World Wide Web Consortium. XML Query Use Cases. *W3C Working Draft*, December 2001.

[12] World Wide Web Consortium. XML Query Requirements. *W3C Working Draft*, February 2001.

[13] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators. *W3C Working Draft*, December 2001.

[14] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. *IETF RFC 2396*.

[15] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. *W3C Recommendation*, October 2000.

[16] World Wide Web Consortium. XML Schema Part 0: Primer. *W3C Recommendation*, May 2001.

[17] P. Brittenham. An Overview of the Web Services Inspection Language, 2001. www.ibm.com/developerworks/webservices/library/ws-wsilover.

[18] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. *IETF RFC 2045*, November 1996.

[19] International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.

[20] Software AG. The Quip XQuery processor. http://www.softwareag.com/developer/quip/.

[21] J. Wang. A survey of web caching schemes for the Internet. *ACM Computer Communication Reviews*, 29(5), October 1999.

[22] S. Gullapalli, K. Czajkowski, C. Kesselman, and S. Fitzgerald. The grid notification framework. Technical report, Grid Forum Working Draft GWD-GIS-019, June 2001. http://www.gridforum.org.

[23] Daniela Florescu, Ioana Manolescu, and Donald Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *Int. Conf. on Very Large Data Bases (VLDB)*, Roma, Italy, September 2001.

[24] Steve Fisher et al. Information and Monitoring (WP3) Architecture Report. Technical report, DataGrid-03-D3.2, January 2001.

[25] W. P. Dinda and B. Plale. A Unified Relational Approach to Grid Information Services. Technical report, Grid Forum Informational Draft GWD-GIS-012-1, February 2001. http://www.gridforum.org.

[26] UDDI Consortium. UDDI: Universal Description, Discovery and Integration. www.uddi.org.

[27] Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual Int. Conf. on Mobile Computing and Networks (MobiCOM '99)*, Seattle, WA, August 1999.

[28] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *IETF RFC 1777*, March 1995.

[29] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Tenth IEEE Int. Symposium on High-Performance Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.

[30] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *6th Int. Symposium on High Performance Distributed Computing (HPDC '97)*, 1997.

[31] The OpenLDAP project. The OpenLDAP project. http://www.openldap.org.

[32] A. Wu, H. Wang, and D. Wilkins. Performance Comparison of Web-To-Database Applications. In *Proceedings of the Southern Conference on Computing*, The University of Southern Mississippi, October 2000.